D S 4

AN INTEGRATED MODELICA ENVIRONMENT FOR MODELING, DOCUMENTATION AND SIMULATION[1]

Peter Fritzson, Vadim Engelson, Johan Gunnarsson

PELAB, Programming Environment Laboratory
Department of Computer and Information Science

Linköping University, S-581 83
Linköping, Sweden

Email: {petfr,vaden,johgu}@ida.liu.se

KEYWORDS

Modelica, Mathematica, programming environments, simulation, documentation

ABSTRACT

Modelica is a new object-oriented multi-domain modeling language based on algebraic and differential equations. In this paper we present an environment that integrates different phases of the Modelica development lifecycle. This is achieved by using the Mathematica environment and its structured documents, "notebooks". Simulation models are represented in the form of structured documents, which integrate source code, documentation and code transformation specifications, as well as providing control over simulation and result visualization.

Import and export of Modelica code between internal structured and external textual representation is supported. Mathematica is an interpreted language, which is suitable as a scripting language for controlling simulation and visualization. Mathematica also supports symbolic transformations on equations and algebraic expressions, which is useful in building mathematical models.

BACKGROUND

Integrated simulation environments are advantageous in order to work effectively and flexibly with simulations. Users prepare and run simulations as well as investigate simulation results. Several auxiliary activities accompany simulation experiments: requirements are specified, models are designed, documentation is associated with appropriate places in the models, input and output data as well as possible constraints on such data are documented and stored together with the simulation model. The user should be able to reproduce experimental results. Therefore input data and parts of output data as well as the experimenter's notes should be stored for future analysis.

Traditionally, simulation and accompanying activities have been expressed using heterogeneous media and tools:

- a simulation model is traditionally designed on paper using traditional mathematical notation;
- simulation programs are written in a low-level programming language and stored on text files;
- input and output data (if stored at all) are saved in proprietary formats needed for particular applications and numerical libraries;
- documentation is written on paper or in separate files that are not integrated with the program files;
- the graphical results are printed on paper or saved using proprietary formats.

When the result of the research and experiments, such as a scientific paper, is written, the user normally gathers together input data, algorithms, output data and its visualizations as well as notes and descriptions. One of the major problem in simulation development environments is that gathering correct versions of all these components from various files and formats is difficult and error-prone.

USING MATHEMATICA NOTEBOOKS

Our approach to the integration problem is based on the Mathematica (Wolfram 1997) environment and its programmable notebooks. Every notebook corresponds to one document (one file) and contains a tree structure of cells. A cell can include other cells and/or arbitrary text or graphics. In particular a cell can include a code fragment or a graph with computational results. The hierarchy of cells corresponds to the traditional hierarchy of chapters, sections, and paragraphs used in textual documents and available in advanced word processors like FrameMaker or MSWord. The text can be written in different styles, including certain font families, bold, italic, color and size as well as mathematical typesetting.

The contents of cells can be

- parts of models (a formal description that can be used for verification, compilation and execution of some simulation model);
- text/documentation (used as comments to executable, formal model specifications);
- dialogue forms for specification and modification of input data;
- result tables (the results can be immediately represented in table form);

1. Published in Proceedings of The 1998 Summer Computer Simulation Conference) (SCSC 98) July 19--22, 1998, Reno, Nevada, pp. 308-313.

- graphical result representation (with 2D vector and raster graphics as well as 3D vector and surface graphics);
- 2D graphs that are used for various model structure visualizations:
  - class diagrams
  - variable dependency diagrams
  - data structure diagrams

THE MODELICA LANGUAGE

The language called Modelica (Modelica 1998) for hierarchical physical modeling has been developed in an international effort. It is an object-oriented language (Elmqvist, 1997, Fritzson and Engelson, 1998) for modeling of physical systems. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a *de facto* standard. It offers three important features: 1) non-causal modeling based on differential and algebraic equations; 2) multidomain modeling capability, i.e. it is possible to combine electrical, mechanical, thermodynamic, hydraulic, etc. model components within the same application model; 3) a general type system that unifies object-orientation, multiple inheritance, and templates within a single `class` construct.

Modelica models are built from classes. Like in other object-oriented languages, a class contains variables, i.e. class attributes representing data. The main difference compared to traditional object-oriented languages is that instead of functions (methods) the programmer uses equations to specify behavior. Equations can be written explicitly, like `a=b`, or can be inherited from other classes. Equations can also be specified by the `connect` statement.

Modelica model of lunar landing

As an introduction to Modelica we will present a model of a rocket landing on the moon surface.

There are two components (*Rocket* and *Planet*) used in the *Landing* class. The class `Rocket` embodies floating point (`Real`) variables, simulation-time constants (`parameter`) and equations for vertical motion of the rocket above the planet surface. The motion is influenced by gravitational force and rocket motor force (`Thrust`). Rocket mass varies since the rocket loses fuel mass proportional to the amount of thrust from the rocket motor. Velocity and acceleration are time-derivatives of height and velocity as specified by the `der(...)` construct.

The class `Landing` includes `apollo` (an instance of class `Rocket`) and `moon` (an instance of `Planet` with specified planet `mass` and `radius`).

Dynamic systems are described by models where behavior evolves as a function of time, i.e. all the variables in the model evolve as functions of time. Modelica has a predefined variable `time` which steps forward during system simulation. The simulation models the rocket behaviour from `time=0` until the rocket touches the ground.

Equations describe the thrust function, a step function with an initial thrust force level `f1` during the time interval [`0..thrustDecreaseTime`] and a second thrust level `f2` during the time interval [`thrustDecreaseTime..thrustEndTime`]. The step functions can conveniently be expressed using Modelica conditional expressions (the **`if-then-else`** construct).

The gravitational field is expressed as:

$$g(t) = \frac{GM_{planet}}{(h(t) + radius_{planet})^2}$$

The Lunar landing model expressed in Modelica

```
class Rocket "generic rocket class"
 Real height(start=59000);
 Real velocity(start=-2000);
 Real mass(start=1000);
 Real acceleration;
 Real Thrust;
 Real gravity;
 parameter Real massLossRate=0.000277;
equation
 Thrust-mass*gravity=mass*acceleration;
 der(mass)=-massLossRate*abs(Thrust);
 der(height)=velocity;
 der(velocity)=acceleration;
end Rocket;
class Planet "generic planet class"
   parameter Real mass;
   parameter Real radius;
end Planet;
class Landing "landing of a rocket onto a
planet"
   parameter Real force1=36000;
   parameter Real force2=1500;
   parameter Real thrustEndTime=210;
   parameter Real thrustDecreaseTime=43;
   parameter Real G=6.672e-11;
   Rocket apollo;
   Planet moon(mass=7.382e22,
               radius=1.738e6);
equation
apollo.Thrust=
 if (time<thrustDecreaseTime) then force1
 else if (time<thrustEndTime) then force2
 else 0;
apollo.gravity=(G*moon.mass)/
         (apollo.height+moon.radius)^^2;
end Landing;
```

Note that Modelica equations do not specify which variables are inputs and which are outputs. Thus, the causality of equations-based models is unspecified. This is fixed only when the equation system is solved.

Simulation Semantics

Classes, instances, and equations are translated by the Modelica compiler into a flat set of differential-algebraic equations, constants and variables. The initial values for `time=0`, specified by the `(start=...)` construct can be taken from the model definition. A simulation engine finds a numerical solution of the system of ordinary differential equations. All model variables are functions of (modeled) time during the simulation.

The simulation engine is a fairly complex piece of software that can be implemented in many different ways. In each case this engine sorts equations (Elmqvist, 1997), finds algebraic loops, checks consistency of ordinary differential equations (ODE), controls an ODE solver, and computes requested variable values from the solution found.

Existing Environment and Need for MathModelica

There is an environment for the Modelica language and engine for Modelica simulations, based on the Dymola system (Dymola 1998). The code, documentation, input data, graphical and numerical results are represented in different, heterogeneous formats, which is a disadvantage of the system. Furthermore, the user is not able to extend the system and introduce new software components. Additionally, the user cannot perform user-specified symbolic (algebraic) manipulations with the formulae used in the code.

For these reasons, we are developing the extensible integrated MathModelica environment which is partly based on the Mathematica system and notebooks.

A specific feature of Mathematica is that models are normally not written as free formatted text. Instead, Mathematica expressions (terms) are used. These can be conveniently written in a tree-like prefix form, or entered using standard mathematical notation. Every term is a number, an identifier or a form such as:

head `[`$term_1$`, ..., `$term_n$`]`

In order to satisfy this requirement, we designed the new *MathModelica language*[1]. Note that MathModelica has the same abstract syntax and the same semantics as Modelica, but a different concrete syntax. This means that essentially the same language constructs are written differently, as illustrated below.

The MathModelica language uses some Mathematica notation, such as:

$term_1$`;...;`$term_n$`, {`$term_1$`,...,`$term_n$`},`
$term_1$ $term_2$, $term_1$`==`$term_2$, `term'`, $term_1$`'`$term_2$,

and arbitrary arithmetic expressions composed from terms.

The ' (tick) means time-derivative. The ` (backtick) character in MathModelica corresponds to . (dot) notation in Modelica.

Lunar Landing Example in MathModelica

```
Class[Rocket;
 Declare[
  Real[start->59000,unit->"m"] height;
  Real[start->-2000,unit->"m/s"] velocity;
  Real[start-> 1000,unit->"kg"] mass;
  Real acceleration;
  Real Thrust;
  Real gravity;
  Parameter Real massLossRate=0.000277 ;
 ];
 Equation[
  Thrust-mass*gravity==mass*acceleration;
  mass'==-massLossRate*Abs[Thrust];
  height'==velocity;
  velocity'==acceleration;
 ]
];

Class [Planet; "**********************"
 Declare[
   Parameter Real mass;
   Parameter Real radius;
 ]
];

BeginClass[Landing];
Declare[
  Parameter Real force1=36000;
  Parameter Real force2=1500;
  Parameter Real thrustEndTime=210;
  Parameter Real thrustDecreaseTime=43;
  Parameter Real G=6.672*10^-11;
  Rocket  apollo;
  Planet[mass->7.382*10^22,
         radius->1.738*10^6]  moon;
];


Equation[
 apollo'Thrust==
  Which[time<thrustDecreaseTime,  force1,
        time<thrustEndTime,      .force2,
        True,                     0];
```

1. This paper presents the preliminary syntax of MathModelica.

```
    apollo'gravity==
                          G·moon'mass
                  (apollo'height+moon'radius)^2
];
EndClass[Landing];
```

Expressions used in MathModelica are "dynamic". This means that they can be created as result of symbolic and algebraic transformation performed by Mathematica. These transformations can be integration, differentiation, expansions of series, simplification, etc.

Formal Syntax of MathModelica

The formal concrete syntax of MathModelica is described by a BNF grammar[1] as shown below, where keywords are all marked with **bold**, other terminals are courier (if they are written as here) or *italic* (generic tokens, like integers or strings):

```
program::= class*
class::= shortClass | longClass
shortClass::= Class [classname;
          (ext | decls | equations )* ];
```

When the short class definition is evaluated in Mathematica, the class description is added to the model.

```
longClass::=BeginClass [classname];
        (ext | decls | equations | comment )*
            EndClass [classname];
```

The constructs **BeginClass[]** and **EndClass[]** are balanced. Class definition can be split, and written in several cells. The **EndClass[***name***];** adds (when evaluated), definition of class name to the syntax tree. These constructs allow the user to insert documentation into class code (Figure 1).

A class can extend (inherit) other classes as in Java:

```
ext ::= Extends [classname*];
decls      ::= Declare [ onedecl* ];
equations ::= Equation[ oneequation* ];
onedecl    ::= [Parameter] type
     [ [ attr *] ] [ [idx *] ]
      varname [ = value ];["short comment";]
```

Note that "parameters" are simulation time constants and their value can be stated in the simulation model.

```
idx ::= integer (used for array declaration)
attr ::=
    start->value; (bound value of variable for time=0)
  |unit->"string" (unit used for user interface only)
  | varname->value; (specifies value for component)
comment::= text written in a separate notebook cell
type ::= Real | Integer | classname
oneequation ::= expr == expr ;
expr ::=  varname' | expr+expr
   | expr-expr | expr/expr | expr*expr
   | builtinfunction(expr) | expr[[expr]]
   | expr'varname | number | expr^expr
```

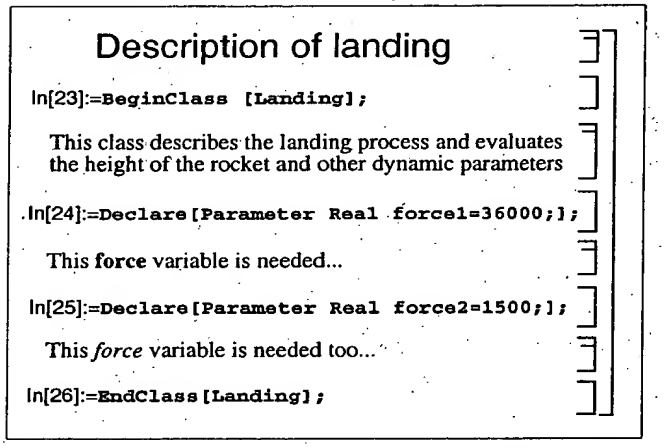

Fig. 1. Notebook fragment with MathModelica class declaration and comments.

In Figure 1 the symbols in the right margin denote different kinds of cells and the hierarchy constructed from the cells. The bracket ⊐ means a cell with code (an executable command); the bracket with the horizontal bar ∃ means a cell with text (a fragment of documentation). A larger ⊐-bracket marks a cell that contains eight smaller cells. A hierarchy of cells of unlimited depth can be easily created.

THE EXPERIMENTAL ENVIRONMENT

MathModelica is both a language and an environment.

The Mathematica built-in functions and MathModelica environment functions can be used in order to manipulate the model. Naturally, the most important operation using the model is simulation. This can be performed in different ways:

- translation of the model to Mathematica, and simulation using the Mathematica built-in ODE solver;
- translation to Modelica, and using the Dymola-based environment for simulation;
- translation to C++ using the MathCode system (MathCode 1998, Fritzson 1997), a Mathematica to C++ compiler.

1. In BNF notation [F] means optional text F; F|G is an alternative between F and G; F* is repetition of F 0 or more times (Aho *et al.* 1986).

TRANSLATION FROM MATHMODELICA TO MATHEMATICA

In order to convert a MathModelica model to Mathematica the function `ModelicaToMathematica[]` is called. Variables and functions declared in the model then become interactively accessible by the user working in the Mathematica environment.

Classes, variables and equations in the MathModelica model are converted to a flat list of variables and equations. First, the constants are set:

```
setparam[]=(apollo`massLossRate = 0.000277;
            thrustDecreaseTime=43;
            thrustEndTime=210;
            moon`mass=7.382*10^22;
            moon`radius=1.738*10^6;
            force1=36000;
            force2=1500;
            G=6.672*10^-11);
```

The Mathematica differential equation solver (`NDSolve`) requires that only differential equations are specified in the equation list, so algebraic equations have to be handled separately.

There is an equation (here named `eq1`) with the variable `apollo`acceleration` the translator has to extract:

```
eq1:=
 apollo`Thrust[t] - apollo`mass[t]*
 apollo`gravity[t]  ==
   apollo`mass[t]*apollo`acceleration[t];
```

To extract the variable `apollo`acceleration` the algebraic equation solver is called:

```
Solve[eq1,apollo`acceleration[t]];
```

The `Solve` function expresses the variable `apollo`acceleration[t]` in terms of other variables. A new function definition for the acceleration is created:

```
apollo`acceleration][t_]:= - ( (
  apollo`gravity[t] * apollo`mass[t] -
   apollo`Thrust[t]) / apollo`mass[t]);
```

Two other functions are already given in the MathModelica code:

```
apollo`gravity[t_]:=(G*moon`mass) /
        (apollo`height[t]+moon`radius)^2;

apollo`Thrust[t_]:=
  If[t < thrustDecreaseTime ,force1,
    If[t < thrustEndTime, force2, 0]];
```

Now, three ordinary differential equations are specified:

```
eqs={
 apollo`mass'[t]==- apollo`massLossRate*
                Abs[ apollo`Thrust[t]],
 apollo`height'[t]==apollo`velocity[t],
 apollo`velocity'[t]==apollo`acceleration[t]
};
```

Initial conditions are necessary for the solution:

```
initcond={
   apollo`height[0]==59000,
   apollo`velocity[0]==-2000,
   apollo`mass [0]==1000};
```

Solutions are required for these functions:

```
vars={apollo`height[t],apollo`velocity[t],
   apollo`mass[t]}
```

When the function `ModelicaSimulate[time`max`]` is called, initial conditions, equations, and functions are passed to an ordinary differential equation solver which is a Mathematica built-in function:

```
res=NDSolve[Join[eqs,initcond],var,
            {t,0,time`max`}];
```

As a result, Mathematica creates polynomial approximations for all functions we want to find, known as interpolate-functions.

Result visualization

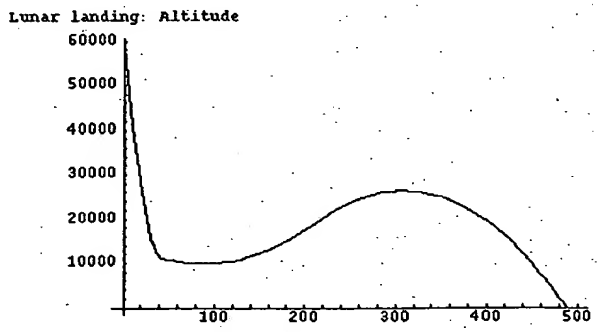

Fig. 2. The `apollo`height` value for time in [0,time$_{max}$] (time$_{max}$=500)

The functions computed by `NDSolve` can be visualized graphically. For instance, the height of the rocket can be visualized (see Figure 2) by the call

```
ModelicaPlot[ apollo`height[t],
              "Lunar landing: Altitude"];
```

Other Scripting Utilities

Three scripting utilities – `ModelicaToMathematica[]`, `ModelicaSimulate[]` and `ModelicaPlot[]` have been illustrated above.

`ImportModelica[]` accepts textual traditional Modelica code, parses it and converts to MathModelica code. `ExportModelica[]` performs the opposite translation. It can be used with a Modelica implementation, available in the Dymola environment.

Input Data Specification

Another script command, `ModelicaInData[]` allows selecting parameter values and specifying boundary conditions interactively. The dialogue window is created automatically from specification of data structures (Engelson and Fritzson, 1996) in the Modelica code (see Figure 3). Variable values can be edited and new simulation runs can produce results immediately. In the future this can be used for computational steering of Modelica simulations.

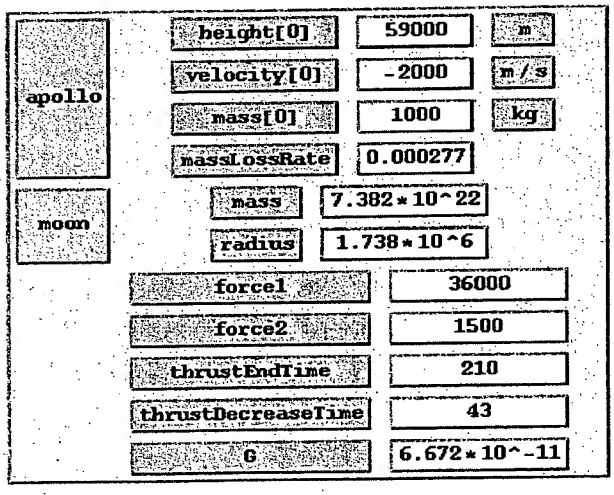

Fig. 3. A Mathematica dialogue box for setting input variable values of a MathModelica simulation. The boxes `apollo` and `moon` represent cell instances with several components.

CONCLUSION

The MathModelica is an extension of the Modelica language targeted for work within the Mathematica environment. This language is object-oriented (the programs consist of collections of classes). The language is equation based: instead of traditional functions and procedures we use non-causal equations which specify algebraic and differential relations between numerical variables. Input-output causality is not specified, and therefore these equations can be used in multiple ways.

The environment integrates most activities needed in simulation design and use: documentation, modeling (coding), symbolic processing and transformation of formulas, input and output data visualization. This advanced programming environment can be applied in various simulation applications.

ACKNOWLEDGMENTS

This work had been supported by the Wallenberg Foundation in the WITAS project.

REFERENCES

Aho, A., Sethi, R., Ullman, J., *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986.

Dymola Home Page, *http://www.Dynasim.se*, 1998.

Elmqvist, H., Mattson, S.E., Modelica - The Next Generation Modeling Language - An International Design Effort, in *Proceedings of First World Congress of System Simulation*, Singapore, September 1-3, 1997.

Engelson, V., Fritzson, P., Fritzson, D., Automatic Generation of User Interfaces From Data Structure Specifications and Object-Oriented Application Models. In *Proceedings of ECOOP96*, Linz, Austria, 8-12 July 1996, Pierre Cointe (Ed.), pp. 114-141. Springer-Verlag, 1996 .

Fritzson, P., Static and Strong Typing for Extended Mathematica. In *Innovation in Mathematics*. Proceedings of the Second International Mathematica Symposium, Rovaniemi, Finland, July 1997, V.Keränen, P. Mitic, A. Hietamäki (Ed.), pp. 153-160.

Fritzson, P. and Engelson, V., Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. Accepted for publication in *Proceedings of ECOOP-98*, Brussels, July 1998.

Fritzson, P., and Fritzson, D., The need for high-level programming support in scientific computing applied to mechanical analysis. *Computers and Structures*, Vol 45, No 2, pp 387-395, 1992.

Modelica Home Page, *http://www.Modelica.org* 1998.

MathCode Home Page, http://www.mathcore.com, 1998

.Wolfram, S., *The Mathematica Book*, Wolfram Media, 1997.